软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(2):195−210 [doi: 10.3724/SP.J.1001.2011.03753]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

扩展 QVT Relations 实现业务流程模型的转换*

何 啸[1,2], 麻志毅[1,2+], 张 岩[1,2], 邵维忠[1,2]

[1](北京大学 信息科学技术学院 软件研究所,北京 100871)

[2](高可信软件技术教育部重点实验室(北京大学),北京 100871)

Extending QVT Relations for Business Process Model Transformation

HE Xiao[1,2], MA Zhi-Yi[1,2+], ZHANG Yan[1,2], SHAO Wei-Zhong[1,2]

[1](Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

+ Corresponding author: E-mail: mzy@sei.pku.edu.cn

He X, Ma ZY, Zhang Y, Shao WZ. Extending QVT Relations for business process model transformation. *Journal of Software*, 2011,22(2):195−210. http://www.jos.org.cn/1000-9825/3753.htm

Abstract: QVT (Query/View/Transformation) Relations cannot describe the pattern that includes other patterns, so it has some difficulties in defining the rules of a process model transformation. To solve this problem, the paper extends QVT Relations with three new concepts: Pattern Factor, Nested Relation Expression, and Relationship End Constraint. The paper also discusses the pattern matching and the model creating semantics after the extension. At Last, a case study is presented to show how the extension could deal with process model transformations.

Key words: MDA (model driven architecture); SOA (service oriented architecture); business process model transformation; QVT (Query/View/Transformation)

摘 要: QVT(Query/View/Transformation) Relations 无法描述包含嵌套模式,因此在描述流程模型的转换规则时存在一些困难.针对此问题,对 QVT Relations 进行了扩展,引入了模式因子、嵌套关系表达式和关联端约束这 3 个概念,并讨论了扩充之后匹配模型和创建模型的语义变化.并用一个例子展示,扩展之后的 QVT Relations 可以解决流程模型转换的问题.

关键词: 模型驱动体系结构;面向服务的体系结构;业务流程模型转换;QVT

中图法分类号: TP311 **文献标识码**: A

模型转换(model transformation)是将一种模型转换成同一系统中另一种模型的过程[1].它是模型驱动体系结构(model driven architecture,简称 MDA)的核心技术之一.近年来,模型驱动的方法成为一种开发 SOA (service oriented architecture)应用的重要方法.建模人员首先利用某一建模语言,例如 UML(unified modeling language)活动图[2]、BPMN(business process modeling notation)[3]等来定义一组平台无关的业务逻辑模型;然后再利用模型转换技术,将这些平台无关模型转换成与特定实现或运行平台相关的模型和代码[4],例如 BPEL (business

* 基金项目: 国家自然科学基金(60773152); 国家重点基础研究发展计划(973)(2005CB321805); 国家高技术研究发展计划(863)(2007AA01Z127, 2007AA010301)

收稿时间: 2008-12-31; 修改时间: 2009-07-06; 定稿时间: 2009-10-09

process executable language)模型[5]或 JAVA 代码.

为了进行流程模型的转换,开发人员需要定义出一组流程模型的转换规则(transformation rule).这些转换规则定义了如何把一个输入模型(input model)转换成一个输出模型(output model).每条转换规则通常由两个模式(pattern)组成,即源模式和目标模式.在应用一条转换规则时,首先需要在输入模型中找到源模式的匹配,并根据目标模式在输出模型中创建出相应的模型片段(根据模式所创建的模型片段也是该模式的匹配).

QVT(Query/View/Transformation) Relations(以下简称 QVT)[6]语言是 OMG(Object Management Group)组织发布的一种转换规则描述语言.然而,流程模型之间的转换规则比较复杂,这些规则中的模式常常存在相互嵌套甚至递归嵌套的情况.所谓模式的嵌套,即在一个模式中包含另一个模式.QVT 虽然是一种规范的转换规则描述语言,但却无法描述相互嵌套的模式.这使得 QVT 无法描述一些流程模型之间的转换规则,从而影响了 QVT 的适用性.

本文针对 QVT 模式描述能力不足这一问题,通过增加模式因子(pattern factor)、嵌套关系表达式(nested relation expression)和关联端点约束(association end constraint)这 3 个新概念对 QVT 语言进行了扩展.其中,模式因子和嵌套关系表达式用于描述基于规则的嵌套模式,关联端点约束用于消除可能的歧义匹配.上述扩展使得 QVT 能够描述嵌套模式,增强了 QVT 的表达能力,并且能够对流程模型的转换规则进行描述.

1 背 景

1.1 QVT Relations的基本概念

QVT Relations 是一种声明式的语言,它可以描述两个 MOF(meta object facility)[7]模型之间的转换规则.QVT 包含文字语法和图形语法,为了便于理解,本文主要使用其图形语法.

一个用 QVT 描述的转换包含若干转换规则,每条规则都称为一个 relation.图 1 是一个用 QVT 图形语法定义的规则,它将一个 UML 类转换成数据库中的表,其左上角的文字“ClassToTable”是这个规则的名字.图中的六边形符号称为关系符号.关系符号两边的图结构是两个模式,左边的是源模式,右边的是目标模式.模式中的结点用矩形图元表示,结点之间的边用实线表示.在进行匹配时,结点用来匹配模型中的元素,边用来匹配关系.

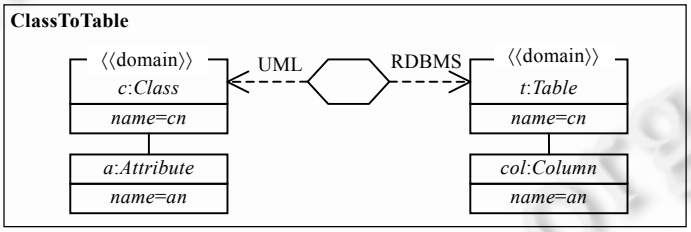


Fig.1 An example of transformation rule

图 1 一个转换规则的例子

结点的表示法可以分为上、下两部分.其中,第 1 栏里的文字定义了这个结点的名字以及这个结点应该匹配的元素类型,如 *c*:*Class* 表明这个结点的名字叫做 *c*,这个结点必须匹配类型为 *Class* 的模型元素.结点图元的第 2 栏中包含一些表达式,这些表达式定义了该结点所对应的模型元素应该满足的一些条件.例如,图 1 中结点 *c* 所包含的表达式 *name=cn*,其中,*name* 是 *Class* 的一个属性,*cn* 则是一个变量.这表明在进行模式匹配时,结点 *c* 必须用来匹配一个类型为 *Class* 且其 *name* 属性的值为 *cn* 的元素.模式中的边表示关联,图 1 中源模式结点 *c* 和结点 *a* 之间存在一条边,它表示 *Class* 和 *Attribute* 之间的聚合关系.源模式表示一个名为 *cn* 的类,且该类至少拥有 1 个名为 *an* 的属性.

每个模式都必须关联一个特定元模型,它的结构必须符合这个元模型的约束.模式所对应的元模型的名字都写在指向该模式的虚线箭头上.例如,图 1 中,关系符号到源模式之间的虚线箭头上的字符串是 UML,这表明源模式是用来匹配 UML 模型的,它所对应的是 UML 元模型.

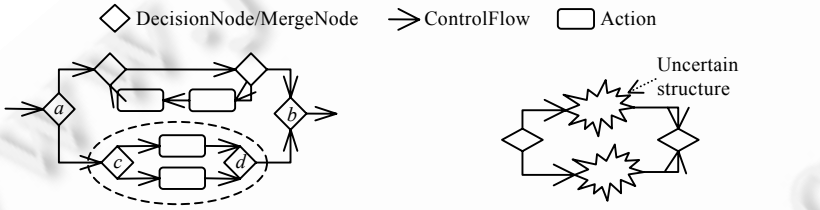
1.2 流程模型转换中的模式嵌套

模式可以分为确定型与非确定型两种.如果该模式的所有匹配都是同构的,那么这个模式就是一个确定型模式;反之,这个模式则是一个非确定型模式.对于流程模型的转换规则来说,其中的模式大多是非确定型模式.例如图 2(a)所示的部分 UML 活动图模型.假设存在一个分支模式 P_{branch},并希望用它来匹配如图 2(a)所示的模型.图 2(a)中虚线框内的结构是一个典型的分支结构,那么图 2(a)中虚线圈出的结构应该是 P_{branch} 的一个匹配.但图 2(a)所示的模型本身也是一个分支结构,因此它也应该是一个 P_{branch} 的匹配.显然,这两个匹配并不同构,由此可以推导出 P_{branch} 不是一个确定型模式.

事实上,分支模式 P_{branch} 可以不严格地描述成如图 2(b)所示的结构.在这个结构中包含一个 DecisionNode(左边的菱形)和一个 MergeNode(右边的菱形),但在这两个元素之间的结构却是不确定的(图中多边形所示的部分).这些不确定的部分既可以表示一个简单的 Action 元素,如图 2(a)中虚线圈出的分支结构,*c* 和 *d* 之间是两个 Action 元素;也可以是其他流程结构,如图 2(a)整体所构成的分支结构中,从 *a* 到 *b* 的两个分支上分别是一个循环结构和一个分支结构.为了描述这样的模式,本文首先定义:

定义 1(嵌套模式). 如果一个模式的定义中包含其他模式,就称这个模式是嵌套模式.若一个嵌套模式的定义中包含它自己,则称该模式是递归嵌套模式.当然,被包含的模式也可以包含其他模式,包含的深度没有限制.

如图 2(a)所示,由于分支结构与其他流程结构发生了嵌套,从而导致分支模式的不确定性.大部分流程模式都是这种包含复杂嵌套的非确定型模式.如图 2(b)所示,为了描述模式 P_{branch},需要将图中不确定的部分定义成“Action 元素或其他模式,也包括 P_{branch} 自己”.也就是说,这两个部分的定义是复杂的结构,而不是简单的元素.但在 QVT 的语法中,模式中的结点只能代表一个简单的模型元素,而不能表示一个结构,更不能表示其他模式.这一缺陷使得它不能定义流程模型的转换规则.



(a) A branch structure (a) 一个分支结构

(b) Uncertain structures in the branch structure (b) 分支结构中的非确定结构

Fig.2

图 2

2 对 QVT 语法的扩充

2.1 模式因子

由于 QVT 无法定义嵌套模式,对此,本文为 QVT 的图形语法增加一个新的符号,名为模式因子(pattern factor).在模式中,模式因子是一种特殊的结点,它可以用来表示一个被嵌套的模式,也可以用来表示一个普通的模型元素.当模式因子用来表示一个普通的模型元素时,它和 QVT 中原有的结点具有相同的功能.

模式因子的图形表示如图 3(a)所示.模式因子是模式中的一个结点,所以它的图形表示和普通的结点符号类似——都是使用矩形表示.其不同在于,模式因子的符号被“〈〈pattern〉〉”修饰.每个模式因子都有一个名字 name,用来与规则中的其他结点相区分,这与普通的结点相同.此外,每个模式因子都有一个候选类型列表 types,用来表示这个模式因子可以匹配哪些元素类型或模式;而普通的结点只能指定一个可以匹配的元素类型.候选类型列表中不同的元素类型名和模式名之间使用“|”来进行分割.候选类型列表中最多包含 1 个元素类型名,但可以包含多个模式名(由于模式因子可以有多个可匹配模式,本文中用“模式因子 *a* 匹配了模式 *P*”的方式来说明

实际匹配的结果符合哪个模式).

图 3(b)是一个模式因子的实例,字符串“*f*:*Action*|*TBranch.UML*”中:“*f*”为该模式因子的名字;“*Action*|*TBranch.UML*”是候选类型列表,里面包含两个候选类型.其中,*Action* 是一个元模型元素的名字,而 *TBranch.UML* 是某一个模式的标识名.这个实例的含义是,定义了一个名叫 *f* 的模式因子,*f* 既可以用来匹配一个类型为 *Action* 的模型元素,也可以用来匹配另一个标识名为 *TBranch.UML* 的模式.

(a) Graphical symbol of pattern factor (b) An example of pattern factor
(a) 模式因子的图形符号 (b) 一个模式因子的例子

Fig.3
图 3

为了指明模式因子可以匹配哪些模式,本文使用模式标识名的概念来指明一个特定的模式.模式的标识名是用来唯一标识一个模式的字符串,它可以定义为“模式所在的规则的名字.模式对应的元模型名”.

按照上述定义,在一组转换规则中,每个模式的模式标识名都是唯一的.首先,QVT 的模式一定是某个规则的源模式或目标模式,而规则的名字是唯一的;其次,一个规则中的任何一个模式只能对应 1 个元模型,一个元模型也只能对应 1 个模式.因此,只要知道模式所在的规则的名字和模式对应的元模型的名字,就可以在一组规则中唯一找到一个模式.例如“TBranch.UML”表示规则 TBranch 中对应元模型为 UML 的模式.

图 4 是一个使用了模式因子描述分支模式的例子,ControlFlow 元素对应图 2(b)中的控制流.图 4 使用了两个模式因子 *a*1 和 *a*2 来描述图 2(b)中不确定的部分.

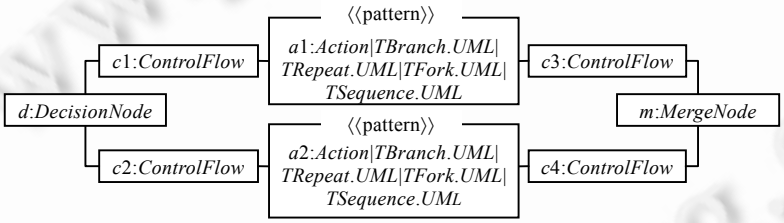


Fig.4 Branch pattern defined by pattern factor
图 4 使用模式因子描述分支模式的例子

在 QVT 中,模式不是孤立的概念,任何一个模式一定是某个规则的源端或目标端.如果有一个模式 P_s,那么一定存在另一个模式 P_t,使得 P_s 和 P_t 构成规则 *Rule*.假设 P_s 是源模式,那么在进行模型转换时,一个 P_s 的匹配会按照 *Rule* 的定义转换成一个 P_t 匹配.在嵌套模式中,类似的关系也是存在的.如图 5 所示,假设 P_s 和 P_t 分别是规则 *Rule* 的源端和目标端,P_s 是嵌套模式,它包含模式 P_{ns}.如果在输入模型中可以找到 P_s 的一个匹配 M_s,作为被嵌套的模式 P_{ns} 必须也存在一个匹配 M_{ns},且 $M_{ns}\subset M_s$.在进行转换时,M_s 应该被转换成目标端的模型 M_t,且 M_t 匹配 P_t.这样,M_s 和 M_t 才能满足规则 *Rule*.由于 P_{ns} 不是独立的模式,必须存在另一个模式 P_{nt} 与之组成规则 $Rule_n$.同时,M_{ns} 是 P_{ns} 的匹配,因此,在将 M_s 转换到 M_t 时,M_{ns} 也必须根据 $Rule_n$ 的定义转换成输出模型 M_{nt},使之匹配 P_{nt}.M_{nt} 作为目标端的模型的一部分,显然包含于 M_t.为了使 M_t 符合 P_t,P_t 中必须包含 $Rule_n$ 的目标端模式 P_{nt}.

由于本文使用模式因子来定义嵌套模式,模式中包含其他模式就意味着在这个模式的定义中包含一个模式因子.根据图 4 所示及上述的说明,本文可以得到两个事实:

(1) 如果源模式中包含一个模式因子 *a*,它可以匹配 P_{ns},那么目标模式中也一定包含一个模式因子 *b*,它可以匹配 P_{nt},并且 P_{ns} 和 P_{nt} 能够组成规则 $Rule_n$.同理,对于目标模式中包含模式因子的情况也类似.这就

是说,不能出现规则的一端包含模式因子而另一端则不包含的情况,或者两端包含的模式因子不能对应起来(两个模式因子对应的条件在第 2.2 节中讨论).

(2) 使用模式因子来定义嵌套模式的方式是一种基于规则嵌套.由于事实(1)的存在,当在规则 *Rule* 的源端和目标端分别嵌套了规则 $Rule_n$ 的源模式和目标模式后,可以看作规则 *Rule* 中嵌套了规则 $Rule_n$.因为在如图 5 所示的转换过程中,为了将 *Rule* 源模式的匹配 M_s 转换成输出模型 M_t,*Rule* 需要调用 $Rule_n$ 才能将 M_{ns} 转换成 M_{nt},从而完成 M_s 到 M_t 的转换.

因此,本文中利用模式因子定义嵌套模式是一种基于规则的嵌套.亦即在模式嵌套的过程中,相应的规则也进行了嵌套.

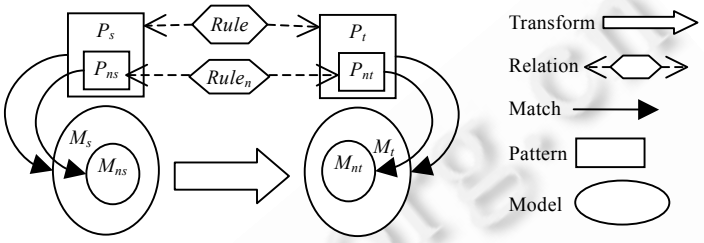


Fig.5 Rule-Based pattern nesting

图 5 基于规则的模式嵌套

2.2 模式因子的对应

如第 2.1 节中所述,模式因子必须成对出现,一个在源端而另一个在目标端,二者相互对应.由于一个规则中可能出现不止 1 对模式因子,为了明确它们之间的对应关系,本文还为 QVT 增加了一个嵌套关系表达式的概念.

嵌套关系表达式的语法为

$$Relation(a,b) \tag{1}$$

其中,*a* 和 *b* 是两个模式因子,*Relation* 是新定义的关键字.

使用嵌套关系表达式描述两个模式因子的关系时,还需要满足一些语法上的要求,即两个模式因子能够对应起来的条件.具体包括如下条件:

(1) *a* 和 *b* 都是模式因子,且 *a* 和 *b* 分别出现在源端模式和目标端模式中;

(2) 如果 *a* 的候选类型列表中包括一个类型名称,那么 *b* 中也必须包括一个类型名,但二者不必相同;反之,如果 *b* 中包含一个类型名,那么 *a* 中也必须包含一个;

(3) 如果 *a* 的候选类型列表中包含一个模式,那么 *b* 中必须包含另一个模式,这两个模式构成一条规则;反之,如果 *b* 中包含一个模式,那么 *a* 中也必须包含另一个模式,这两个模式构成一条规则.

对于一个嵌套模式中的所有模式因子,都必须使用嵌套关系表达式将其两两对应起来.嵌套关系表达式出现在一个 QVT 规则的 where 子句中.

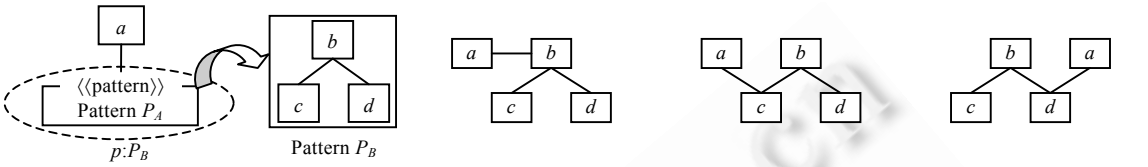
嵌套关系表达式是一种特殊的关系调用表达式(CallRelationExpression).普通的关系调用表达式主要用来计算某些模型元素是否满足特定的规则.例如,一个关系调用表达式为 *ClassToTable*(*c*,*t*),计算这个表达式就是计算结点 *c* 匹配的模型元素和结点 *t* 匹配的模型元素是否满足规则 *ClassToTable*.嵌套关系表达式则是计算模式因子的匹配是否能够满足一条规则.它与普通的关系调用表达式的区别在于,所验证的规则是在运行时决定的——依赖于模式因子所匹配的模式.因此,对于表达式 *Relation*(*a*,*b*),如果在运行时模式因子 *a* 和 *b* 分别匹配了某个规则 *Rule* 中的两个模式,计算这个表达式就意味着验证 *a* 和 *b* 的匹配是否满足规则 *Rule*.

2.3 关联端点约束

引入模式因子虽然可以定义嵌套模式,但也会带来一些副作用,这就是可能导致匹配时的歧义问题.这主要

是因为模式因子可以表示一个模式(是一个复杂结构),而与之相连的边则可能以该模式中的任何一个结点作为端点;由于一个模式可能包含多个结点,因此在匹配时会出现歧义.

图 6 是一个匹配发生歧义的例子.图 6(a)中存在两个模式:模式 P_A 中包含一个模式因子 p 和一个普通的结点 a,a 和 p 之间存在一条边;模式因子 p 可以匹配另一个模式 P_B,模式 P_B 中包含 3 个结点 b,c 和 d.在匹配 P_A 时,a 和 p 之间的边存在 3 种可能的匹配,即 a 和 b 之间的关联,如图 6(b)所示;a 和 c 之间的关联,如图 6(c)所示;a 和 d 之间的关联,如图 6(d)所示.



(a) Patterns P_A and P_B (a) 模式 P_A 和 P_B (b) The 1st match instance (b) 第 1 种匹配实例 (c) The 2nd match instance (c) 第 2 种匹配实例 (d) The 3rd match instance (d) 第 3 种匹配实例

Fig.6

图 6

为此,本文引入关联端点约束(association end constraint)的概念来消除匹配歧义.关联端点约束是一种特殊的约束,当一条边和一个模式因子相连时,关联端点约束指明了这个模式因子所代表的模式中有哪个结点可以与这条边相连.关联端点约束必须出现在某条边的一端,并且这一端连接的结点必须是模式因子.其语法是

⟨*PatternIdentifier*:*NodeName*⟩.

如图 7 所示,模式中的边在靠近模式因子 p 的一端增加了一个关联端点约束⟨P_B:b⟩,这表明这条边只能匹配结点 a 和模式 P_B 中的结点 b 之间的关联.此时,可能的情况只有图 6(a)这一种情况.如果模式因子 p 可以用来匹配多个模式,比如模式 P_B 和模式 P_C,此时必须分别消除这个关联针对模式 P_B 和模式 P_C 的匹配歧义,则关联端点约束可以写成⟨P_B:b⟩⟨P_C:x⟩.其中,⟨P_B:b⟩是针对模式 P_B 的标记,⟨P_C:x⟩是针对模式 P_C 的标记(假设 P_C 中存在一个结点 x).这个标记表示,当模式因子 p 匹配了模式 P_B 时,这个关联只能表示 a 和 b 之间的关联;但模式因子 p 匹配了模式 P_C 时,这个关联只能表示 a 和 x 之间的关联.

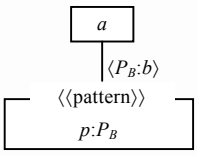


Fig.7 Association end constraint

图 7 关联端约束

2.4 对QVT元模型的扩充

在 QVT 的元模型中,一个 Pattern 由一组 Variable 和一组 Predicate 组成.在图形语法中,模式里的每个元素都对应一个 Variable,模式中表达式和约束都对应 Predicate.本文第 2.1 节所作的扩充——为 QVT 的图形语法增加模式因子,实际上是定义了一种新的 Variable.而本文第 2.3 节中增加的关联端点约束,它用于对模式进行约束,因此它是一种新的 Predicate.本文第 2.2 节扩充的嵌套关系表达式定义了两个模式因子必须要满足的对应关系,因此它是一种特殊的关系调用表达式.

如图 8 所示的元模型是在 QVTBase 包和 QVTRelation 包[6]中针对 QVT 语言的元模型所作的扩展.在如图 8 所示的元模型结构中,用灰色背景的元模型元素及它们之间的关系都是 QVT 规范中定义的原有结构,用白色背景的元模型元素及其关系都是本文对 QVT 元模型的扩充.

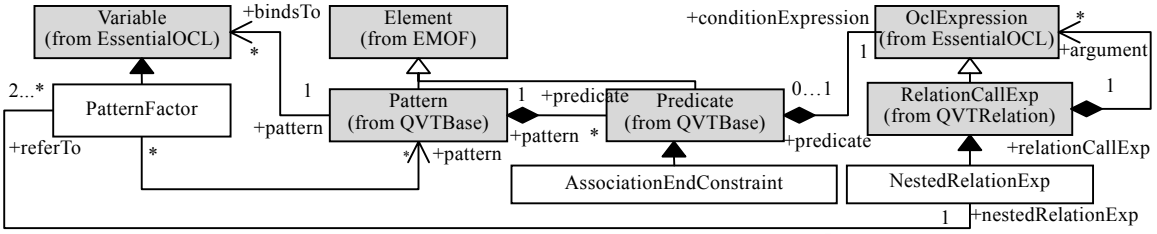


Fig.8 Extended metamodel of QVT

图 8 扩展的 QVT 元模型

PatternFactor 用来表示本文中扩充的模式因子.在 QVT 中,模式中的每个结点都是一个 Variable(在 EssentialOCL 包[8]中定义)元素.Variable 是一个有名字、有类型的元素[6],Variable 的名字就是结点的名称,其类型则是这个结点可以匹配的模型元素类型;在运行时,Variable 的取值就是这个结点的一个匹配.由于模式因子是一种特殊的结点,因此,PatternFactor 是 Variable 的一个子类.其次,模式因子的候选类型列表中可以包含多个模式标识名,每个模式标识名又代表了一个模式,因此,在 PatternFactor 和 Pattern 之间存在一个关联关系.又由于每个模式标识名也可以出现在多个模式因子的候选类型列表中,因此,PatternFactor 和 Pattern 之间的关联关系是一个多对多的关系.

嵌套关系表达式在图 8 中使用 NestedRelationExp 元素来表示,它是一种特殊的关系调用表达式,即 NestedRelationExp 是 RelationCallExp 的特殊类.NestedRelationExp 和 PatternFactor 之间存在一对多的关系.该关系表明,嵌套关系表达式可以将两个或两个以上的模式因子对应起来(本文只讨论了两个的情况),但每个模式因子只能出现在 1 个嵌套关系表达式中.

最后,AssociationEndConstraint 用来表示一个关联端点约束.由于每个关联端点约束实际上就是一个约束条件,它刻画了模式匹配时需满足的一些额外条件,从而消除歧义匹配.因此,AssociationEndConstraint 是 Predicate 的一个特殊类.

3 语 义

对于一个规则 *Rule*,假设 *Rule* 的源模式是 P_s,目标模式是 P_t,那么在输入模型 M_{input} 上应用规则 *Rule*,从而得到输出模型 M_{output} 的过程,即 *Rule* 执行转换时的语义可以简化地表述为:

(a) 在 M_{input} 进行模式匹配,找到 P_s 的所有候选匹配所构成的集合记为$|P_s|$;

(b) 对于$|P_s|$中的每个匹配实例 M_s;

(c) 根据 P_t 的定义和 *Rule* 中约束,创建 M_s 的输出模型 M_t;

(d) 令 $M_{output}=M_{output}\cup M_t$.

出于简化的考虑,上述过程中没有考虑 QVT 中 when 子句和 where 子句的相关语义.如前所述,when 子句可以理解为规则的前置条件集合,而 where 子句可以理解为规则的后置条件集合.对于本文的扩展,只有嵌套关系表达式可能出现在 where 子句中.但这并没有改变 when 子句和 where 子句本身的语义,而只是新增了嵌套关系表达式的语义.这将在第 3.3 节中给出.

在本节中,一个模式 P 可以定义为一个三元组,即 $P=\langle O,L,Pred\rangle$.

$O=\{o_1,o_2,\ldots,o_n\}$表示 P 中的结点集合,其中的每个元素也可以用元组的形式表示.对于任意 $o\in O$,如果 o 是一个普通的结点,那么 o 定义为$\langle name,type\rangle$,其中,*name* 表示这个结点的名字,*type* 表示这个结点可以匹配的元素类型.如果 o 是一个模式因子,那么可以定义 $o=\langle name,type,PaL\rangle$,其中,*name* 和 *type* 的含义不变,$PaL=\{P_{o1},P_{o2},\ldots,P_{ok}\}$是 o 可以匹配模式的集合,*type* 和 *PaL* 一起构成了第 2.1 节中引入的候选类型列表.模式因子的 *type* 可以为空(*null*).例如,图 3(b)中的模式因子可以表示成$\langle f,Action,\{TBranch.UML\}\rangle$.

$L=\{l_1,l_2,\ldots,l_m\}$表示 P 的边集合.对于任意 $l\in L$,定义 $l=\langle o_i,o_j\rangle$,其中,$o_i\in O$ 并且 $o_j\in O$.

$Pred=\{pr_1,pr_2,\ldots,pr_l\}$是这个模式中的约束表达式集合.通常,$pr_i$是一个OCL[8]表达式,由相应的OCL引擎负责计算.在增加了第2.3节中的扩展之后,$Pred$中可能包含关联端点约束.关联端点约束是一种约束,它约束了与模式因子相连的边应如何进行匹配.

关联端点约束可以表示成$\langle l,o,\{\langle P_{o1},o_{p1}\rangle,\langle P_{o2},o_{p2}\rangle,\ldots,\langle P_{ov},o_{pv}\rangle\}\rangle$.其中,$l\in L$,表示模式中的一条边;$o\in O$,$o$必须是一个模式因子,并且$o$必须是$l$的一个端点.$P_{ox}$表示某个模式,$P_{ox}\in o.PaL$,$1\leqslant x\leqslant v$;$o_{px}\in P_{ox}.O$,$1\leqslant x\leqslant v$.例如,图7中的关联端点约束$\langle P_B:a\rangle$可以表示成$\langle l_1,p,\{\langle P_B,a\rangle\}\rangle$,其中,$l_1=\langle a,p\rangle$.如果这个关联端点约束是$\langle P_B:b\rangle\langle P_C:x\rangle$,那么这个约束是$\langle l_1,p,\{\langle P_B,b\rangle,\langle P_C,x\rangle\}\rangle$.

一条转换规则$Rule$可以定义为$Rule=\langle P_s,P_t,when,where\rangle$,其中,$P_s$是源模式,$P_t$是目标模式,$when$是QVT中的$when$子句,$where$是QVT中的$where$子句.由于每个模式都一定属于一条规则,对于一个模式P,本节定义$P\rightarrow rule()$表示P所属的规则,即$P\rightarrow rule()=Rule_p$,其中,$Rule_p=\langle P,P_t,when,where\rangle$或者$Rule_p=\langle P_s,P,when,where\rangle$.即$P$是$Rule_p$的一个模式.

嵌套关系表达式主要出现在$where$子句中,一个嵌套关系表达式可以定义为一个二元组$rel=\langle o_s,o_t\rangle$,其中,$o_s\in P_s.O$,$o_t\in P_t.O$,且$o_s$和$o_t$都是模式因子.

M表示模型,可以将M定义为$M=\langle E,R\rangle$.

$E=\{e_1,e_2,\ldots,e_u\}$表示模型中所有的元素集合.对任意$e\in E$,e都有一个类型,本节定义$e\rightarrow type()$表示元素e的类型.

$R=\{r_1,r_2,\ldots,r_v\}$表示模型中所有的关系集合.对任意$r\in R$,$r=\langle e_p,e_q\rangle$,其中,$e_p\in E$并且$e_q\in E$.

符号说明:对于一个特定的元组,本文使用“.”操作符来表示元组中的分量.例如,对于一个模式P_1,$P_1.L$表示模式P_1中的边集.

3.1 模式匹配语义

由于篇幅的限制,本节无法给出完整的嵌套模式匹配的算法,而是给出了其对应的判定问题的算法,即如何判断一个模型结构匹配某个嵌套模式的条件.理论上,匹配的算法可以通过枚举所有的子图,并反复检查这个判断条件,从而找到所有的匹配.虽然这样的实现效率很低,但是:(1) 子图匹配问题本身是NP完全问题,所有解法效率都不高;(2) 本节的目的在于讨论匹配时的语义,而不关心算法的效率.

考虑一个模型M_{input}以及M_{input}的一个子图M_{sub},其中,$M_{sub}.E=\{e_1,e_2,\ldots,e_m\}$,且$M_{sub}.E\subseteq M_{input}.E$,$M_{sub}.R\subseteq M_{input}.R$.对于某个模式$P$,其中,$P.O=\{o_1,o_2,\ldots,o_n\}$,$P.L=\{l_1,l_2,\ldots,l_k\}$,那么$M_{sub}$是$P$的一个匹配,记作$P\sim M_{sub}$,需要依次满足以下条件:

(1) 划分可满足:存在至少1种划分,能够将$e_1,e_2,\ldots,e_m$分成$g_1=\{e_1,\ldots,e_i\}$,$g_2=\{e_{i+1},\ldots,e_j\}$,$\ldots$,$g_n=\{e_{k+1},\ldots,e_m\}$这$n$个组.其中,$|g_i|\geqslant 1$,$1\leqslant i\leqslant n$,即不能有空分组.

(2) 结点可匹配:P中的每个结点都是可匹配的.对于所有的$0\leqslant i\leqslant n$,o_i是可匹配的当且仅当以下条件之一成立,此时,称g_i匹配o_i,记作$o_i\sim g_i$:

  (a) 无论o_i是普通的结点还是模式因子,即$o_i=\langle name,type\rangle$或$o_i=\langle name,type,PaL\rangle$:如果$g_i=\{e_t\}$,并且$e_t\rightarrow type()=o_i.type$,或$e_t\rightarrow type()$是$o_i.type$的一个子孙类,则称$o_i$是可匹配的.

  (b) 若o_i是一个模式因子,即$o_i=\langle name,type,PaL\rangle$:考虑$g_i$中元素在$M_{sub}$中构成的子图$M'_{sub}$,即$M'_{sub}=(g_i\cap M_{sub})$.如果存在某个$P_o\in o_i.PaL$,使得$P_o\sim M'_{sub}$成立,则称$o_i$是可匹配的.

(3) 边可匹配:对于所有的$0\leqslant i\leqslant k$,$l_i=\langle o_{i1},o_{i2}\rangle$在$M_{sub}$中有匹配,若$o_{i1}\sim g_{i1}$,$o_{i2}\sim g_{i2}$.$l_i$在$M_{sub}$中有匹配,当且仅当存在$e_{i1}\in g_{i1}$和$e_{i2}\in g_{i2}$,并且$\langle e_{i1},e_{i2}\rangle\in M_{sub}.R$,此时记作$l_i\sim\langle e_{i1},e_{i2}\rangle$.在这一步中无须检查关联端点约束的约束,这些约束将在最后一步进行检查.

(4) 约束可满足:即当元素匹配和关联匹配满足的情况下,约束表达式都为真.在QVT中,普通的约束表达式都是一些OCL表达式,这些表达式由专门的OCL引擎负责计算.关联端点约束是一种特殊的约束,如果一个关联端点约束是$\langle l,o,\{\langle P_{o1},o_{p1}\rangle,\langle P_{o2},o_{p2}\rangle,\ldots,\langle P_{ov},o_{pv}\rangle\}\rangle$,且$l\sim\langle e_i,e_j\rangle$,$o\sim g$,其中,$g\in\{g_1,g_2,\ldots,g_n\}$.这个关联端点约束约束的成立条件是以下条件之一:

(a) 若 $g=\{e_i\}$,即 o 匹配了$\langle e_i,e_j\rangle$的一个端点 e_i,则关联端点约束计算结果为真.

(b) 与条件(a)类似,若 $g=\{e_j\}$,即 o 匹配了$\langle e_i,e_j\rangle$的另一个端点 e_j,结果为真.

(c) 若$\{e_i\}\subset g$,则存在一个$\langle P_x,o_y\rangle\in\{\langle P_{o1},o_{p1}\rangle,\langle P_{o2},o_{j2}\rangle,\ldots,\langle P_{ov},o_{pv}\rangle\}$,使得 $P_x\sim(g\cap M_{sub})$,并且在$(g\cap M_{sub})$中,$o_y\sim\{e_i\}$.

(d) 与条件(c)类似,若$\{e_j\}\subset g$,则存在一个$\langle P_x,o_y\rangle\in\{\langle P_{o1},o_{p1}\rangle,\langle P_{o2},o_{p2}\rangle,\ldots,\langle P_{ov},o_{pv}\rangle\}$,使得 $P_x\sim(g\cap M_{sub})$,并且在$(g\cap M_{sub})$中,$o_y\sim\{e_j\}$.

从上面说明的模型匹配条件可以看出,这个条件是递归定义的.在条件(2)的分支条件(b)中出现了递归.由于在增加了模式因子以后允许在一个模式中嵌套其他模式,甚至是递归嵌套,因此在匹配时,匹配的过程也就出现了递归.关联端点约束作为约束在条件(4)中检查,但这是一种逻辑上的顺序,在实现时,可以根据情况在匹配边时(条件(3))中)进行检查.

3.2 模型创建的语义

目标端的模式在模型转换过程中被用作创建模型的模版,目标端的模型将根据目标端模式来进行创建.创建的过程只是根据模式中描述的结构以及属性设置的方法实例化新的模型.

创建模型的过程可以简单地分成两个部分:(1) 根据模式中的结点创建模型元素;(2) 根据模式中的边创建关系.在 QVT 中定义了一个操作 *CreateOrUpdate*(o,M),其中,o 是模式中的一个结点,M 是一个模型.这个操作的作用是在 M 中根据 o 的定义创建或更新一个模型元素;同样地,如果要创建或更新一个关系,在本文中使用 *CreateOrUpdate*(l,e_1,e_2,M)的方式表示,其中,l 是模式中的一条边,e_1 和 e_2 都是 M 中的元素.如果要创建的元素在 M 中已经存在,则执行更新操作,否则,执行新建操作[6].关系的创建也类似.本节中的创建操作未加说明时均指 *CreateOrUpdate* 这种方式的操作.

对于一个规则 $Rule=\langle P_s,P_t,when,where\rangle$,其源模式是 P_s,目标模式是 P_t.如果输入模型是 M_{input},输出模型是 M_{output},那么对于 P_s 在 M_{input} 中的一个匹配 M_s,即 $P_s\sim M_s$,在 M_{output} 中创建模型的操作记作 *CreateByPattern*(P_t,M_s,M_{output}).这个操作的主要步骤如下:

(1) 创建元素.对于 P_t 中的所有结点,即对任意 $o_t\in P_t.O$,在 M_{output} 中创建元素.这一步又可以分成两种情况:

(a) 如果 o_t 是一个普通的结点,即 $o_t=\langle name,type\rangle$,那么根据 o_t 的定义,调用 *CreateOrUpdate*(o_t,M_{output})创建新的元素.设新元素是 e_t,显然 $o_t\sim\{e_t\}$.

(b) 如果 o_t 是一个模式因子,即 $o_t=\langle name,type,PaL\rangle$,对于嵌套关系表达式 $rel=\langle o_s,o_t\rangle\in P_t\rightarrow rule()$. *where*,$o_s$ 是 P_s 中的模式因子.考虑 o_s 在 M_s 中的匹配 g_s,即 $o_s\sim g_s$:

i. 如果 o_s 在 M_s 中匹配了一个普通的模型元素,设 $g_s=\{e_s\}$,那么把 o_t 强制转换成一个普通的结点,再调用 *CreateOrUpdate*(o_t,M_{output})创建新的元素.设新元素是 e_t,则 $o_t\sim\{e_t\}$.

ii. 如果 o_s 匹配了一个模式 P_{os},其中,$P_{os}\in o_s.PaL$,即 P_{os} 是 o_s 候选类型列表中的一个模式,设 $g_s=\{e_{s1},e_{s2},\ldots,e_{su}\}$,使得 $P_{os}\sim(g_s\cap M_s)$,那么一定存在一个 $P_{ot}\in o_t.PaL$,并且存在规则 $Rule_o=\langle P_{os},P_{ot},when,where\rangle$,即 P_{os} 和 P_{ot} 来自于同一条规则.此时调用 *CreateByPattern*(P_{ot},$(g_s\cap M_s)$,M_{output})创建相应的模型结构.创建的结果显然是一个模型结构,设这个结构是 M_{ot},那么显然有 $P_{ot}\sim M_{ot}$ 且 $o_t\sim M_{ot}.O$.

(2) 创建关系.对于 P_t 中的所有边,即对任意 $l_t=\langle o_{t1},o_{t2}\rangle\in P_t.L$,在 M_{output} 中创建相应的关系.这一步的关键是根据关联端点约束中的约束设置正确的起点和终点.假设在步骤(1),在元素创建后,设 $o_{t1}\sim g_{t1}$,并且 $o_{t2}\sim g_{t2}$,创建关系的情况可以分成以下几种情况:

(a) 如果 $g_{t1}=\{e_i\}$,并且 $g_{t2}=\{e_j\}$,即在创建元素时,执行引擎根据 o_{t1} 和 o_{t2} 都只创建了 1 个模型元素,则 *CreateOrUpdate*(l_t,e_i,e_j,M_{output}).

(b) $g_{t1}=\{e_{i1},e_{i2},\ldots,e_{ik}\}$,并且 $g_{t2}=\{e_j\}$,即 o_{t1} 是一个模式因子.如果存在对于关联 l_t 的关联端点约束$\langle l_t,o_{t1},\{\langle P_{p1},o_{p1}\rangle,\langle P_{p2},o_{p2}\rangle,\ldots,\langle P_{pu},o_{pu}\rangle\}\rangle$,并且存在$\langle P_x,o_y\rangle\in\{\langle P_{p1},o_{p1}\rangle,\langle P_{p2},o_{p2}\rangle,\ldots,\langle P_{pu},o_{pu}\rangle\}$,使得 $P_x\sim(g_{t1}\cap M_{output})$,$o_y\sim\{e_y\}$,其中,$e_y\in g_{t1}$,那么 *CreateOrUpdate*($l_t$,$e_y$,$e_j$,$M_{output}$);如果没有相应的关联端点约束,则任选 $e_i\in g_{t1}$,执行 *CreateOrUpdate*(l_t,e_i,e_j,M_{output}).

(c) $g_{t1}=\{e_i\}$,并且 $g_{t2}=\{e_{j1},e_{j2},\ldots,e_{jl}\}$,即 o_{t2} 是一个模式因子.如果存在关联端点约束$\langle l_t,o_{t2},\{\langle P_{q1},o_{q1}\rangle,\langle P_{q2},o_{q2}\rangle,\ldots,\langle P_{qv},o_{qv}\rangle\}\rangle$,并且存在$\langle P_x,o_y\rangle\in\{\langle P_{q1},o_{q1}\rangle,\langle P_{q2},o_{q2}\rangle,\ldots,\langle P_{qv},o_{qv}\rangle\}$,使得 $P_x\sim(g_{t2}\cap M_{output})$,$o_y\sim\{e_y\}$,其中,$e_y\in g_{t2}$,那么 $CreateOrUpdate(l_t,e_i,e_y,M_{output})$;如果没有相应的关联端点约束,则任选 $e_j\in g_{t2}$,执行 $CreateOrUpdate(l_t,e_i,e_j,M_{output})$.

(d) $g_{t1}=\{e_{i1},e_{i2},\ldots,e_{ik}\}$,并且 $g_{t2}=\{e_{j1},e_{j2},\ldots,e_{jl}\}$.若存在关联端点约束$\langle l_t,o_{t1},\{\langle P_{p1},o_{p1}\rangle,\langle P_{p2},o_{p2}\rangle,\ldots,\langle P_{pu},o_{pu}\rangle\}\rangle$,并且存在$\langle P_{x1},o_{y1}\rangle\in\{\langle P_{p1},o_{p1}\rangle,\langle P_{p2},o_{p2}\rangle,\ldots,\langle P_{pu},o_{pu}\rangle\}$,使得 $P_{x1}\sim(g_{t1}\cap M_{output})$,$o_{y1}\sim\{e_{y1}\}$,那么,令 $e_1=e_{y1}$;否则,令 e_1 是 g_{t1} 中任意的一个元素.若存在关联端点约束$\langle l_t,o_{t2},\{\langle P_{q1},o_{q1}\rangle,\langle P_{q2},o_{q2}\rangle,\ldots,\langle P_{qv},o_{qv}\rangle\}\rangle$,并且存在$\langle P_{x2},o_{y2}\rangle\in\{\langle P_{q1},o_{q1}\rangle,\langle P_{q2},o_{q2}\rangle,\ldots,\langle P_{qv},o_{qv}\rangle\}$,使得 $P_{x2}\sim(g_{t2}\cap M_{output})$,$o_{y2}\sim\{e_{y2}\}$,那么,令 $e_2=e_{y2}$;否则,令 e_2 是 g_{t2} 中任意的一个元素.执行 $CreateOrUpdate(l_t,e_1,e_2,M_{output})$.

从上面的说明可以看出,创建模型的语义也是递归定义的.这是因为增加了模式因子后,模型创建的过程出现了嵌套.

3.3 嵌套关系表达式的语义

在 where 子句执行时,引擎需要检查其中的每个表达式是否成立.经过本文的扩充,其中的表达式可能包括普通的 OCL 表达式、关系调用表达式或嵌套关系表达式.普通的 OCL 表达式仍由 OCL 引擎负责计算,关系调用表达式则根据标准的 QVT 语义进行计算.下面定义嵌套关系表达式的语义.

设输入模型是 M_{input},输出模型是 M_{output},一条规则 $Rule$ 包含一个嵌套关系表达式$\langle o_s,o_t\rangle$,其中,$\langle o_s,o_t\rangle\in Rule.where$,$o_s\in Rule.P_s$,$o_t\in Rule.P_t$,即 o_s 和 o_t 分别是该规则源模式和目标模式中的模式因子.设 $o_s\sim g_s$,$o_t\sim g_t$,其中,$g_s\subseteq M_{input}.E$,且 $g_t\subseteq M_{output}.E$.嵌套关系表达式成立的条件是下面二者之一:

(1) o_s 匹配了一个普通的模型元素,即 $g_s=\{e_s\}$,且 $e_s\rightarrow type()=o_s.type$ 或 $e_s\rightarrow type()$是 $o_s.type$ 的子孙类.同时,o_t 也匹配了一个普通的模型元素,即 $g_t=\{e_t\}$,且 $e_t\rightarrow type()=o_t.type$ 或 $e_t\rightarrow type()$是 $o_t.type$ 的子孙类.此时,嵌套关系表达式为真.

(2) o_s 匹配了模式 P_s,其中,$P_s\in o_s.PaL$,且 $P_s\sim M_s$,其中,$M_s=(g_s\cap M_{input})$.同时,o_t 匹配了模式 P_t,其中,$P_t\in o_t.PaL$,且 $P_t\sim M_t$,其中,$M_t=(g_t\cap M_{output})$.并且 P_s 和 P_t 来自同一个规则 $Rule_n$,即 $P_s\rightarrow rule()=P_t\rightarrow rule()=Rule_n$.并且 M_s 和 M_t 可以满足规则 $Rule_n$ 中定义的所有条件.这一检查过程相当于一个标准的关系调用表达式,即将 M_s 和 M_t 作为输入和输出模型,并调用 $Rule_n$ 以验证该规则是否能够在这两个模型上成立.此时,嵌套关系表达式为真.

4 案例研究

本节将结合一个案例展示如何利用本文给出的扩展 QVT 定义流程模型的转换规则.假设如图 9 所示的 UML 活动图模型是一个 Web Service 的业务逻辑模型模型,希望将其转换成一个用 BPEL 描述的流程模型.在图 9 中,这个 Web Service 首先接受一个价格查询列表(receive price query list),然后依次查询列表中的每个项目.对于每个项目,Web Service 同时向两个供货商 *A* 和 *B* 查询其价格(query from supplier *A* 和 query from supplier *B*),然后比较它们的大小并选择价格低的供货商(select price *A* 或 select price *B*).当所有的查询项处理完毕后,将所有的结果汇总最后返回给用户(output result).

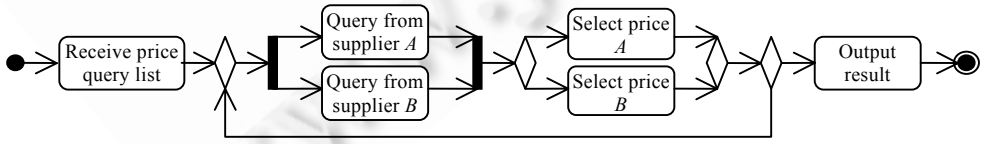


Fig.9 A price query business process model

图 9 查询价格的业务流程模型

通过分析不难发现,图 9 中包含几个基本的流程结构,如图 10 所示.其中包括流程结构(图 10(a))、顺序结构(图 10(b))、选择结构(图 10(c))、循环结构(图 10(d))和并发结构(图 10(e)).本文将根据这几种结构描述模型转换规则.根据图 10 中的流程结构,利用本文扩充后的 QVT 语法定义转换规则如图 11~图 15 所示.

如图 11 所示的规则 TFlow 定义了如何将一个 UML Activity Diagram 的流程转换成的 BPEL 流程.如图 10(a)所示,一个基本的流程就是由一个 *InitialNode* 元素、一个 *FinalNode* 元素和一个活动组成,其中的活动可以是简单的 *Action* 元素,也可以是其他控制结构.因此在转换规则中,中间的活动使用一个模式因子 *a*1 来表示,*a*1 既可以是一个 *Action*,也可以是其他模式,如 *TRepeat.UML*,*TBranch.UML*.在转换的目标,也就是 BPEL 端,一个流程是一个 *process* 元素.按照 BPEL 规范的规定,*process* 元素中只能包含 1 个子元素来表示流程的内容,因此, BPEL 端的模式中使用一个模式因子 *b*1 来表示 *process* 元素的子元素.为了避免歧义匹配,在这个规则中还使用了 3 个关联端点约束,即 *c*1 和 *a*1 之间、*a*1 和 *c*2 之间以及 *p* 和 *b*1 之间(关联端点约束中出现的结点名称都定义在相应的规则中,如⟨*TSequence.UML*:*a*7⟩中的 *a*7 定义在图 15 中).在 where 子句中使用了嵌套关系表达式,定义了 *a*1 和 *b*1 之间的对应关系.

如图 12 所示的规则 TRepeat 是一个将循环结构转换成 BPEL 中 *repeatUntil* 元素的转换规则.在 UML 端的模式中,循环结构中的循环体使用一个模式因子 *a*2 表示.转换的目标端是一个 *repeatUntil* 元素,*repeatUntil* 元素只有 1 个子元素用来表示需要循环执行的动作,因此,在目标端使用模式因子 *b*2 来表示这个循环体,*a*2 会被转换成 *b*2.

如图 13 所示的规则 TBranch 是一个将选择结构转换成 BPEL 中 *if* 元素的转换规则.源端模式表示了一个包含两个分支的选择结构,分支上的活动分别使用两个模式因子 *a*3 和 *a*4 表示,对应的目标模式的模式因子是 *b*3 和 *b*4.

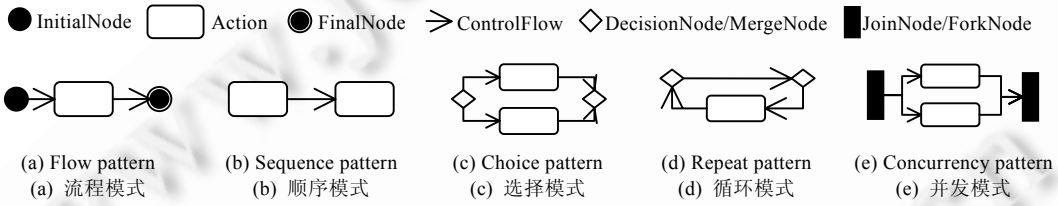


Fig.10

图 10

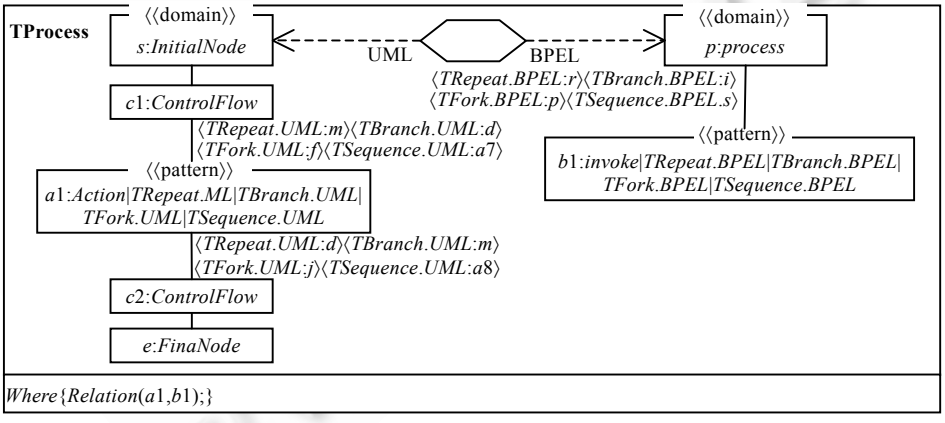


Fig.11 Transformation rule for the flow pattern

图 11 流程模式的转换规则

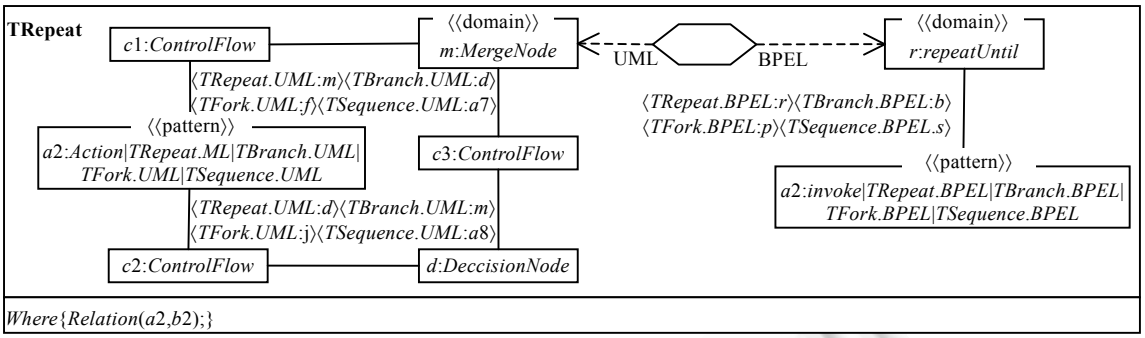


Fig.12 Ttransformation rule for the repeat pattern

图 12 循环模式的转换规则

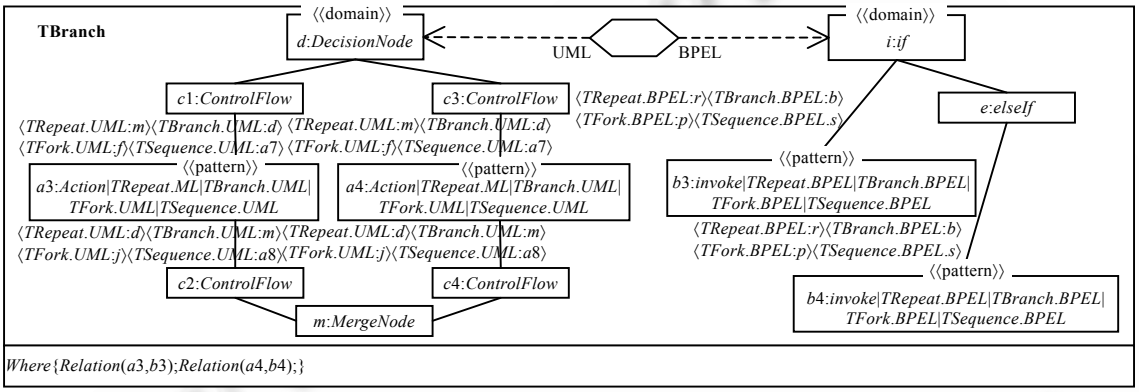


Fig.13 Transformation rule for the choice pattern

图 13 选择模式的转换规则

如图 14 所示的规则 TFork 是一个将并发结构转换成 BPEL 的 flow 元素的转换规则.

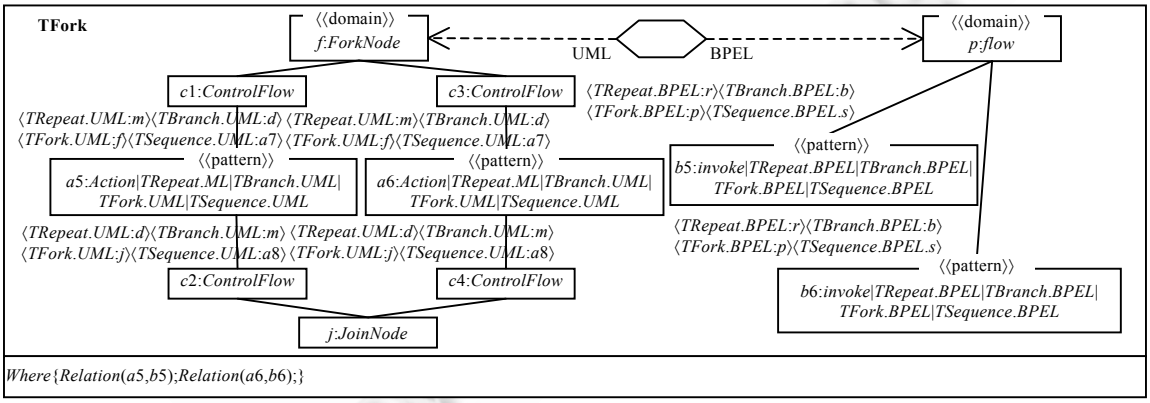


Fig.14 Transformation rule for the concurrent pattern

图 14 并发模式的转换规则

如图 15 所示,规则 TSequence 将顺序结构转换成 BPEL 的 sequence 元素.

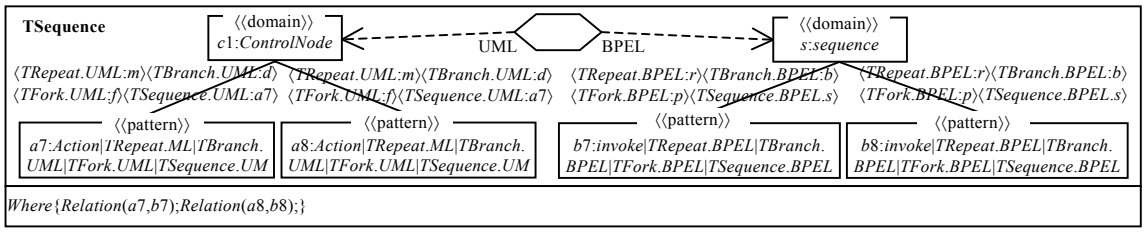


Fig.15 Transformation rule for the sequence pattern

图 15 顺序模式的转换规则

将上面定义的转换规则应用到如图 9 所示的流程模型上,可以得到如图 16 所示的 BPEL 流程模型.在转换过程中,首先应用规则 TProcess,按照第 3 节定义的语义可以逐步得到图 16 的模型.可以看出,本文的扩充使得 QVT 能够定义和处理流程模型之间的模型转换.

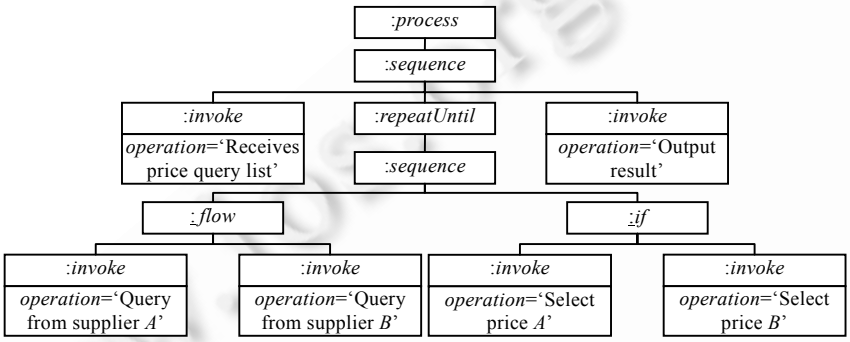


Fig.16 BPEL model after transformation

图 16 转换之后的 BPEL 模型

5 相关工作

与本文相关的工作,首先是一些直接研究如何进行流程模型转换的工作.

Murzek 等人在文献[9]中提出了一种基于模式的业务流程转换方法,在这一方法中,模式被用作沟通不同流程模式的桥梁.该方法首先定义了一些共有的流程模式,然后在不同的模型中分别定义这些模式的具体结构,即将具体的模型结构映射到共有的流程模式上.在转换时,首先根据预先定义的模式将源端模型分解一些基本元素,然后将分解好的元素转换为对应的模型元素,最后再将这些元素组合起来构成输出模型.使用文献[9]中的做法进行流程模型的转换比较简单,因为它只需要开发人员描述不同的模型到共有模式的映射,而无须定义复杂的转换规则.但这要求所转换的模型在结构上必须具有相似性,否则就很难将其映射到同样的流程模式上.因此,这样的做法主要适用于处理水平方向的模型转换——即抽象级别相同的模型的转换,因为同一级别的流程模型结构往往相似.文献[9]的方法不适用于垂直方向的模型转换,特别是 PIM(platform independent model)到 PSM(platform specific model)的转换.而本文针对 QVT 的扩充并不关心被转换的模型是否具有相似的结构,只要能够定义出相应的转换规则就可以用来处理水平和垂直方向的模型转换.例如,本文的案例研究中定义的转换就是一个垂直方向的转换,所转换的模型是异构的.

Ouyang 等人在文献[10]中提出了一种将 BPMN 流程转换成 BPEL 流程的方法,并且这种方法也可以用来将 UML Activity Diagram、工作流图等常见的流程模型转换成 BPEL 流程,并且可以将非结构化的流程转换成 BPEL 流程.该方法主要使用了 BPEL 中的 OnEvent 元素,将流程中由控制流所定义的动作之间的顺序关系转换成 BPEL 活动(activity)之间的事件广播与监听.文献[10]中方法的优点是可以转换任意拓扑结构业务流程模型

到BPEL模型.但是,Chun Ouyang的方法并不是通用的流程模型转换方法,因为这个方法主要用到了BPEL中特殊OnEvent的元素,这就限定了转换的目标只能是BPEL.而本文提出的方法是通过扩展模型转换的描述语言来达到实现流程模型的转换,这种转换并不依赖于任何特殊元素.与文献[10]的方法相比,本文的方法还可以用来处理目标模型不是BPEL的流程转换,具有通用性的特点.魏明等人的工作[4]也是基于BPEL的,它增强了从BPMN模型到BPEL模型转换的能力,从而支持和增强了事务处理和错误处理的转换.在文献[4]中所新定义的几种用于转换规则,也都可以使用本文的扩展进行描述.

Zhao等人在文献[11]中提出了一种将状态机图(state machine diagram)转换成结构化的编程语言的方法.该方法首先将状态机图转换成正则表达式,再通过分析这个正则表达式找出其中的各种顺序、选择、循环等结构,从而将正则表达式转换成一个只包含while,if,switch-case这3种结构化的控制语句,且不含goto语句的源代码.在这之后,他们又指出这一方法可以扩展成将流程图模型转换成BPEL模型的方法.这首先需要将流程图转换成一个抽象的有向图,再将这个有向图看作是一个状态机图,并将其转换成结构化的源代码.由于BPEL模型也是结构化的模型,因此,这个源代码可以比较容易地转换成BPEL模型.但是,该方法无法表示成一组模型转换规则,只能通过复杂的算法实现.转换的需求一旦发生变化,就不得不直接修改转换的算法.而本文的方法是基于转换规则的,当需求发生变化时,只需修改与之有关的规则,其他规则可以保持不变.此外,在文献[11]的方法中,利用状态机图和正则表达式来进行转换,但是这两种工具都无法处理如并发、异常、事件、消息等建模元素.而本文的方法则可以处理这些元素,例如,如图14所示就是一条处理并发结构的规则.对于其他的元素,只需定义相应的规则就可以进行转换了.

还有一些在源代码中消除goto语句的方法[12,13]也可以用来转换流程模型,但是这类方法并没有考虑并发结构.

与本文相关的第2部分工作就是研究模式和模型转换语言的工作.

Wahler在其博士学位论文[14]中介绍了一种利用组合模式定义约束的方法.在该方法中,模式是可以进行组合(嵌套)的,即一个模式可以包含其他模式.利用这样的技术,可以定义出更加复杂的模式,以描述复杂的模型约束.但Wahler的工作主要用于定义模型约束,文献[14]中并没有定义如何利用嵌套模式进行模型的创建及转换.而本文则讨论了如何利用嵌套模式匹配和创建业务流程模型,并定义了相应的语义.

Agrawa在文献[15]中介绍了一种模型转换技术GREAT.GREAT使用势(cardinality)的概念描述具有多重性的模式.文献[15]中的势可以直观地描述重复出现的结点、边,甚至是重复出现的子图,从而能够描述一些不确定型模式.文献[15]的工作与本文的差别在于,文献[15]中的势只能描述多重性,但无法描述模式之间的嵌套;而本文的扩充则主要用于处理嵌套模式.此外,Agrawa在文献[16]中介绍了一种基于控制流的模式嵌套方法.通过使用图形化的符号去描述嵌套的控制逻辑,GREAT也可以支持嵌套模式.但文献[16]与本文最大的不同在于,文献[16]中的方法是一种命令式(imperative)的方法,整个嵌套过程需要编写特殊的控制逻辑;而本文的方法是声明式的(declarative),用户可以像定义普通的模式一样定义嵌套模式.

PROGRES[17]和Fujaba[18]支持一种特殊的概念——Path Expression,这一概念也可以用来定义某些嵌套模式.Path Expression可以定义一条任意长度的路径,路径上的结构可以重复和嵌套.但由于Path Expresion定义了一条路径,因此它只能处理线性结构的嵌套和递归.而本文的方法则还可以用来定义非线性的嵌套模式.

文献[19]提出了一种基于VIATRA的模式组合方法,能够以声明式的方式定义嵌套模式.用户可以在VIATRA中预先定义一些独立的模式,然后在定义转换规则的源模式和目标模式中调用这些模式,从而使得源模式和目标模式中嵌套了这些模式.这些被嵌套的模式可以是任何复杂的结构,它们本身也可以嵌套或递归嵌套模式.文献[20]讨论了VIATRA上嵌套模式的匹配方法.文献[19,20]与本文的工作虽然相似,但其不同在于:文献[19,20]中被嵌套的模式都是单独定义的,不属于任何的规则,所以文献[19,20]中的方法是单纯的模式嵌套,模式在嵌套的同时不附加任何的约束或动作,因此在每次使用嵌套模式时都需要重新定义被嵌套的模式之间的对应关系、转换的方法、参数的传递等;而本文则要求所有的模式都必须属于某个规则,使用嵌套关系表达式来描述模式因子之间的对应关系,执行时可以自动地选择相应的规则进行转换.因此,本文的方法是一种基于转

换规则的模式嵌套,即在描述嵌套模式的同时,实际上也将相应的语义、约束、转换操作进行了嵌套.

6 结论与展望

流程模型转换是一种很重要的模型转换,与静态结构模型转换相比,流程模型转换更加复杂.由于模式描述能力的不足,QVT Relations 作为 OMG 定义的一种转换规则描述语言,不能有效地定义流程模型的转换规则.本文在 QVT Relations 图形语法的基础上进行扩充,增加了一种能够处理模式嵌套的元素——模式因子、嵌套关系表达式与关联端点约束,增强了 QVT Relations 的表达能力.本文还介绍了对 QVT 元模型的扩充,并定义了扩展之后的 QVT Relations 的语义.本文的扩充首先增强了 QVT Relations 的表达能力,使其能够定义流程模型的转换规则.其次,本文讨论的基于规则的模式嵌套的方法,在嵌套模式的同时嵌套相应的转换规则,提高基于规则的模型转换的能力.最后,使用扩充之后的 QVT Relations 实现业务流程模型的转换,进一步提升了 SOA 系统开发的自动化程度.

下一步的研究将继续完善针对 QVT Relations 的扩展.利用关联端点约束来解决匹配歧义的问题只是一种轻量级的方法,虽然关联端点约束可以转换成 OCL 表达式,但是,并非所有的 OCL 表达式都能表示成关联端点约束.因此,对于更加复杂的匹配歧义问题目前还只能使用 OCL 表达式处理.因此,我们下一步将继续研究如何利用图形化的方式处理这一问题.

References:

[1] OMG. MDA guide. Version 1.0.1, OMG, 2003. http://www.omg.org/docs/omg/03-06-01.pdf

[2] OMG. Unified modeling language superstructure. Version 2.1.2, OMG, 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/

[3] OMG. Business process modeling notation. Version 1.1, OMG, 2008. http://www.bpmn.org/Documents/BPMN 1-1 Specification.pdf

[4] Wei M, Xia YL, Wei J. Model transformation from BPMN to BPEL2.0. Application Research of Computers, 2008,25(11): 3363−3366 (in Chinese with English abstract).

[5] Jordan D, Evdemon J. Web services business process execution language version. Version 2.0, Oasis, 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

[6] OMG. Meta object facility (MOF) 2.0 query/view/transformation specification. Version 1.0, OMG, 2008. http://www.omg.org/spec/QVT/1.0/PDF/

[7] OMG. Meta object facility (MOF) core specification. Version 2.0, OMG, 2006. http://www.omg.org/spec/MOF/2.0/PDF/

[8] OMG. Object constraint language specification. Version 2.0, OMG, 2006. http://www.omg.org/spec/OCL/2.0/PDF

[9] Murzek M, Kramler G, Michlmayr E. Structural patterns for the transformation of business process models. In: Proc. of the 10th IEEE Int'l Enterprise Distributed Object Computing Conf. Los Alamitos: IEEE Computer Society, 2006. 18−27. [doi: 10.1109/EDOCW.2006.64]

[10] Ouyang C, Dumas M, Breutel S, ter Hofstede A. Translating standard process models to BPEL. In: Goos G, ed. Proc. of the 18th Int'l Conf. on Advanced Information Systems Engineering. Berlin: Springer-Verlag, 2006. 417−432.

[11] Zhao W, Bryant BR, Cao F, Bhattacharya K, Hauser R. Transforming business process models: Enabling programming at a higher level. In: Proc. of the 2005 IEEE Int'l Conf. on Services Computing. Los Alamitos: IEEE Computer Society, 2005. 173−180. [doi: 10.1109/SCC.2005. 102]

[12] Koehler J, Hauser R. Untangling unstructured cyclic flows—A solution based on continuations. In: Meersman R, Tari Z, eds. Proc. of the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE. Berlin: Springer-Verlag, 2004. 121−138. [doi: 10.1007/978-3-540-30468-5_10]

[13] Zhang FB, D'Hollander EH. Using hammock graphs to structure programs. IEEE Trans. on Software Engineering, 2004,30(4): 231−245. [doi: 10.1109/TSE.2004.1274043]

[14] Wahler M. Using patterns to develop consistent design constraints [Ph.D. Thesis]. Zurich: The Degree of Doctor of Sciences, Swiss Federal Insittute of Technology, 2008.

[15] Agrawal A, Karsai G, Shi F. Graph transformations on domain-specific models. 2003. http://www.escherinstitute.org/tools/suites/mic/great/GReAT_TechnicalReport.pdf

[16] Agrawal A, Vizhanyo A, Kalmar Z, Shi F, Narayanan A, Karsai G. Reusable idioms and patterns in graph transformation languages. In: Mens T, Schurr A, Taentzer G, eds. Proc. of the Int'l Workshop on Graph-Based Tools. San Francisco: Elsevier, 2004. 181−192. [doi: 10.1016/j.entcs.2004.12.035]

[17] Zundorf A. Graph pattern matching in PROGRES. In: Cuny J, Ehrig H, eds. Proc. of the Graph Grammars and Their Application to Computer Science. Berlin: Springer-Verlag, 1996. 454−468. [doi: 10.1007/3-540-61228-9_105]

[18] Nickel P, Niere P, Zundorf A. The FUJABA environment. In: Ghezzi C, Jazayeri M, eds. Proc. of the 22nd Int'l Conf. on Software Engineering. New York: ACM Press, 2000. 742−745. [doi: 10.1145/337180.337620]

[19] Balogh A, Varro D. Pattern composition in graph transformation rules. Technical Report, TR-CTIT-06-34, Enschede: Centre for Telematics and Information Technology, University of Twente, 2006.

[20] Varro G, Horvath A, Varro D. Recursive graph pattern matching with magic sets and global search plans. In: Schurr A, Nagl M, Zundorf A, eds. Proc. of the Applications of Graph Transformations with Industrial Relevance. Berlin: Springer-Verlag, 2007. 456−470.

附中文参考文献:

[4] 魏明,夏永霖,魏峻.BPMN 到 BPEL2.0 的模型转换方法.计算机应用研究,2008,25(11):3363−3366.

何啸(1983－),男,北京人,博士生,主要研究领域为面向对象建模,元建模,模型驱动的体系结构,模型转换技术.

张岩(1974－),男,博士,讲师,主要研究领域为软件工程,形式化方法,模型驱动开发方法,软件度量.

麻志毅(1963－),男,博士,副教授,主要研究领域为软件工程与支撑环境,软件建模技术,面向对象技术.

邵维忠(1946－),男,教授,博士生导师,主要研究领域为软件工程环境,面向对象方法,软件复用,软件构件技术.